A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

Cray MPT:
MPI on the Cray XT

Justin L. Whitt
jwhitt@utk.edu
Glenn Brook
glenn-brook@tennessee.edu
Mark Fahey, Group Leader
mfahey@utk.edu

NICS Scientific Computing Group

OLCF Spring '11
Oak Ridge, TN
March 7 – 11, 2011

Introduction

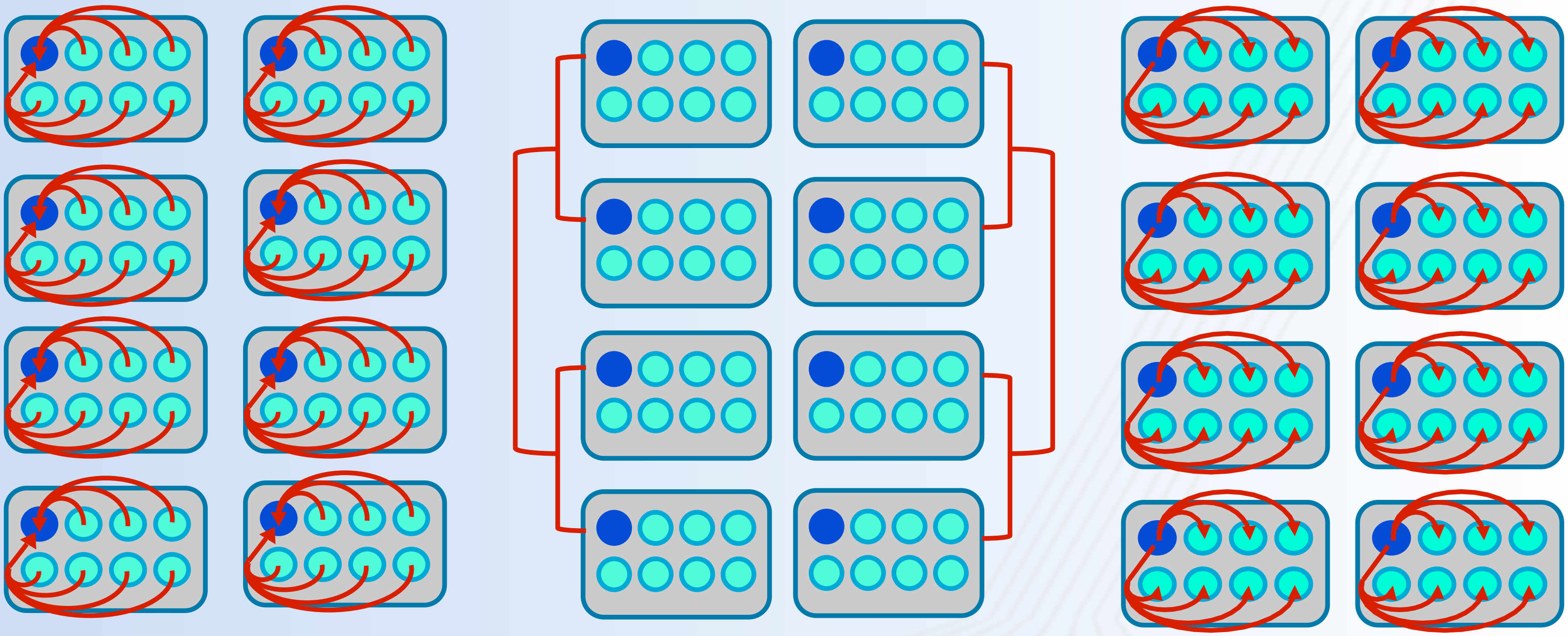
Cray MPT – Message Passing Toolkit

- Cray's MPI library (and SHMEM library)
  - Optimized MPICH-2 for Cray interconnects
- Multiple interconnect devices
  - SMP – Shared memory communication on nodes
  - Portals – Efficient message passing between nodes
- Multiple message protocols
  - Short messages: eager protocol
  - Long messages: rendezvous protocol (default), eager protocol
- Optimized collective communication algorithms
- Automatic transitions between devices, protocols, and algorithms (configurable via environment variables)

Cray Collective Communications

- Improved performance over standard MPICH2
- Work for any communicator (not just MPI_COMM_WORLD)
- User-adjustable thresholds for algorithm selection
- Cray Optimized Collectives
  - MPI_Allgather (small messages) & MPI_Allgatherv
  - MPI_Alltoall (optimized exchange order)
  - MPI_Alltoallv / MPI_Alltoallw (windowing algorithm)
- Cray Optimized SMP-aware Collectives: MPI_Allreduce, MPI_Barrier, MPI_Bcast, MPI_Reduce
- Are enabled by default but can be selectively disabled via MPICH_COLL_OPT_OFF

SMP-aware Collectives – Allreduce Example



STEP 1

Identify Node-Captain rank. Perform a local on-node reduction to node-captain. NO network traffic.

STEP 2

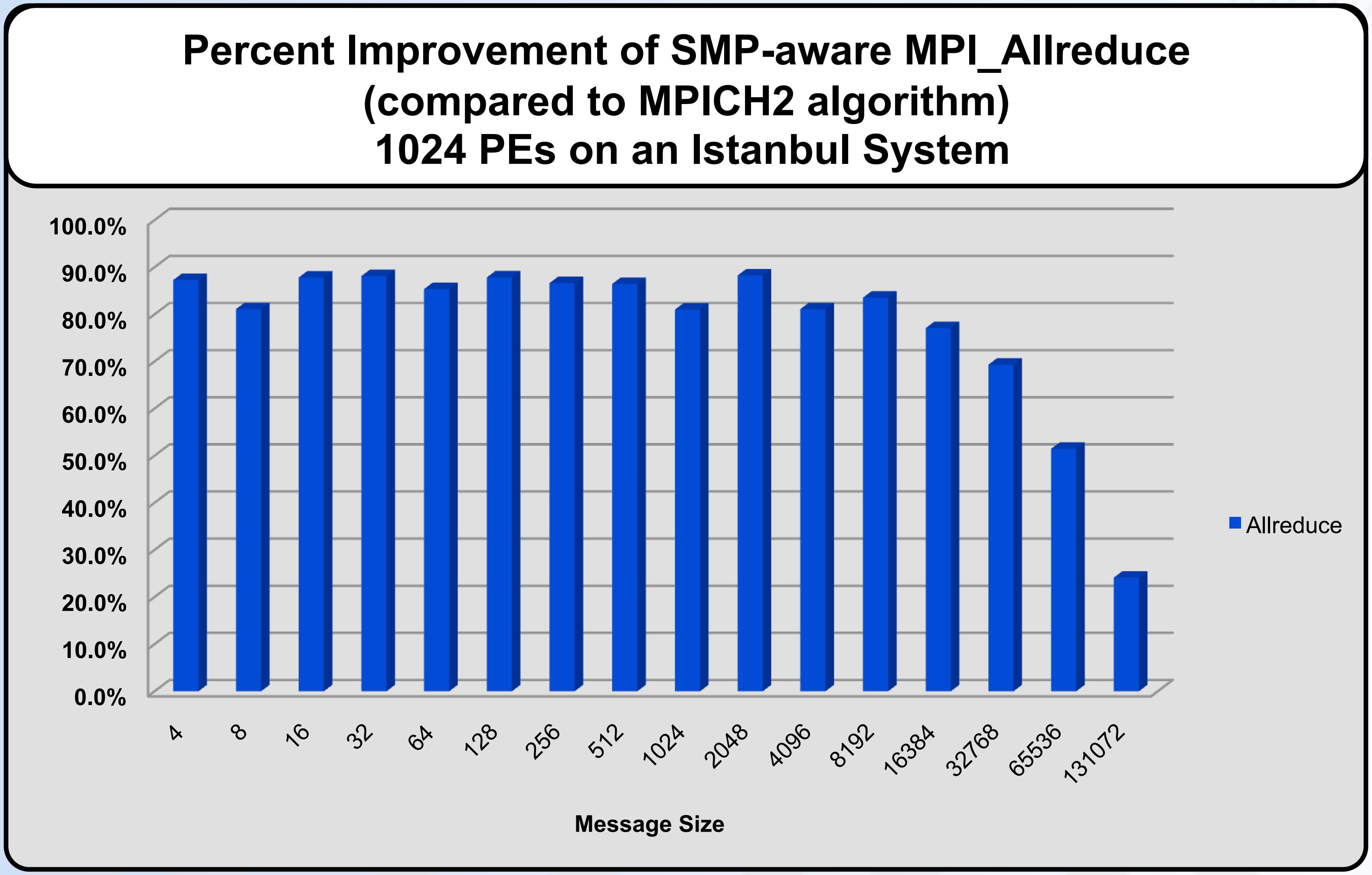
Perform an Allreduce with node -captains only. This reduces the process count by a factor of 8 on XT5.

STEP 3

Perform a local on-node bcast. NO network traffic.

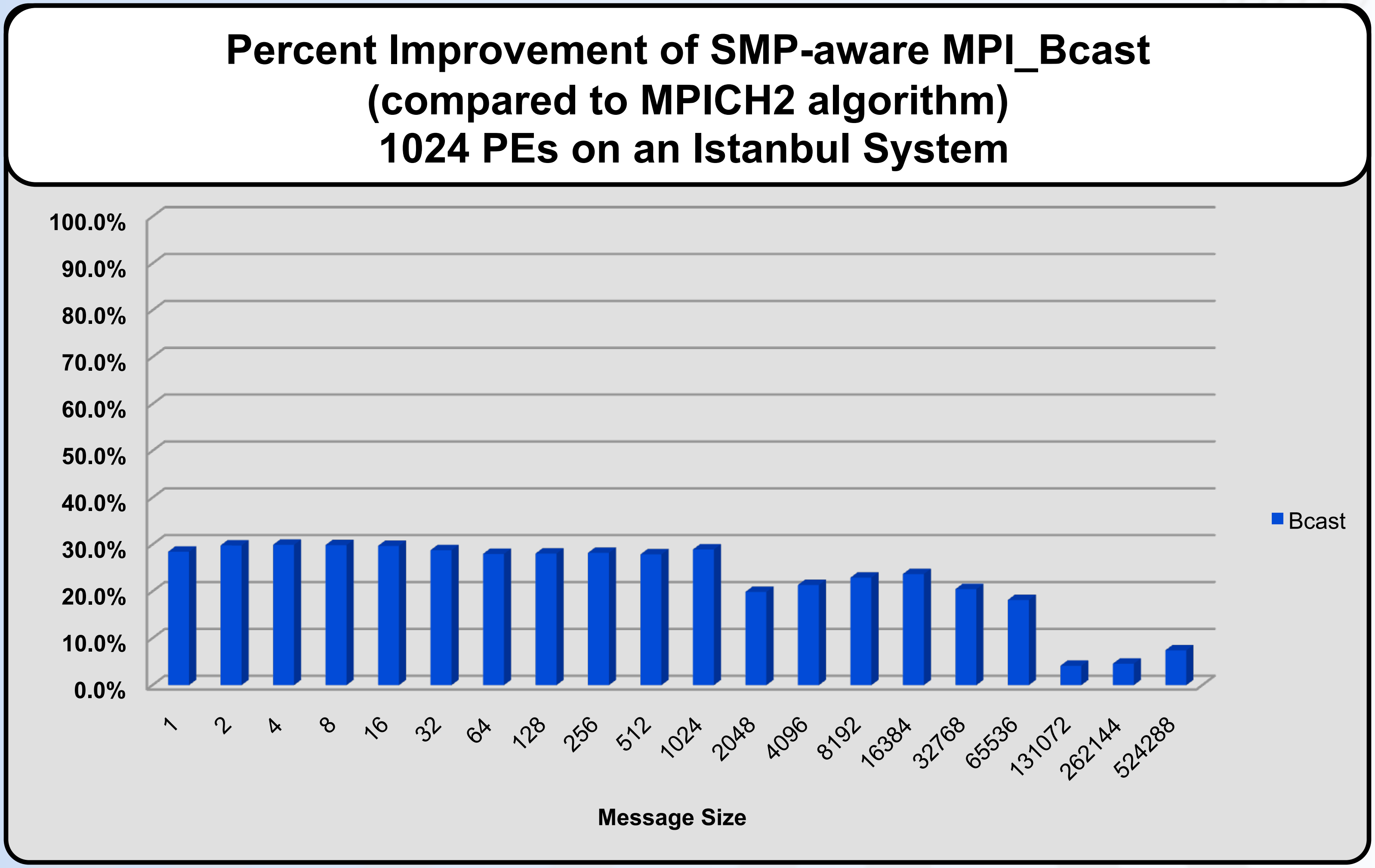
Performance Comparison of MPI_Allreduce

Default vs MPICH_COLL_OPT_OFF=MPI_Allreduce



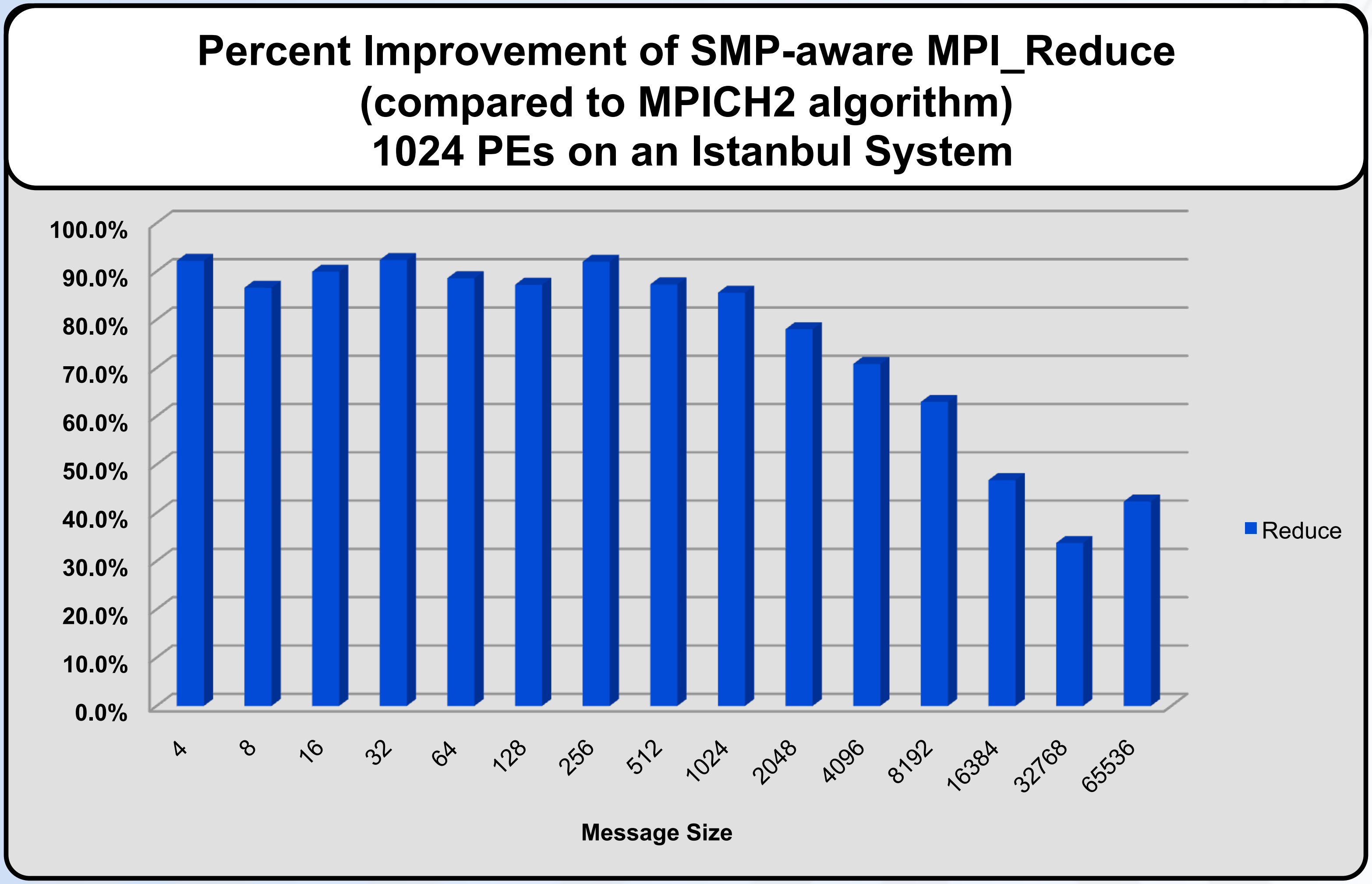
Performance Comparison of MPI_Bcast

Default vs MPICH_COLL_OPT_OFF=MPI_Bcast



Performance Comparison of MPI_Reduce

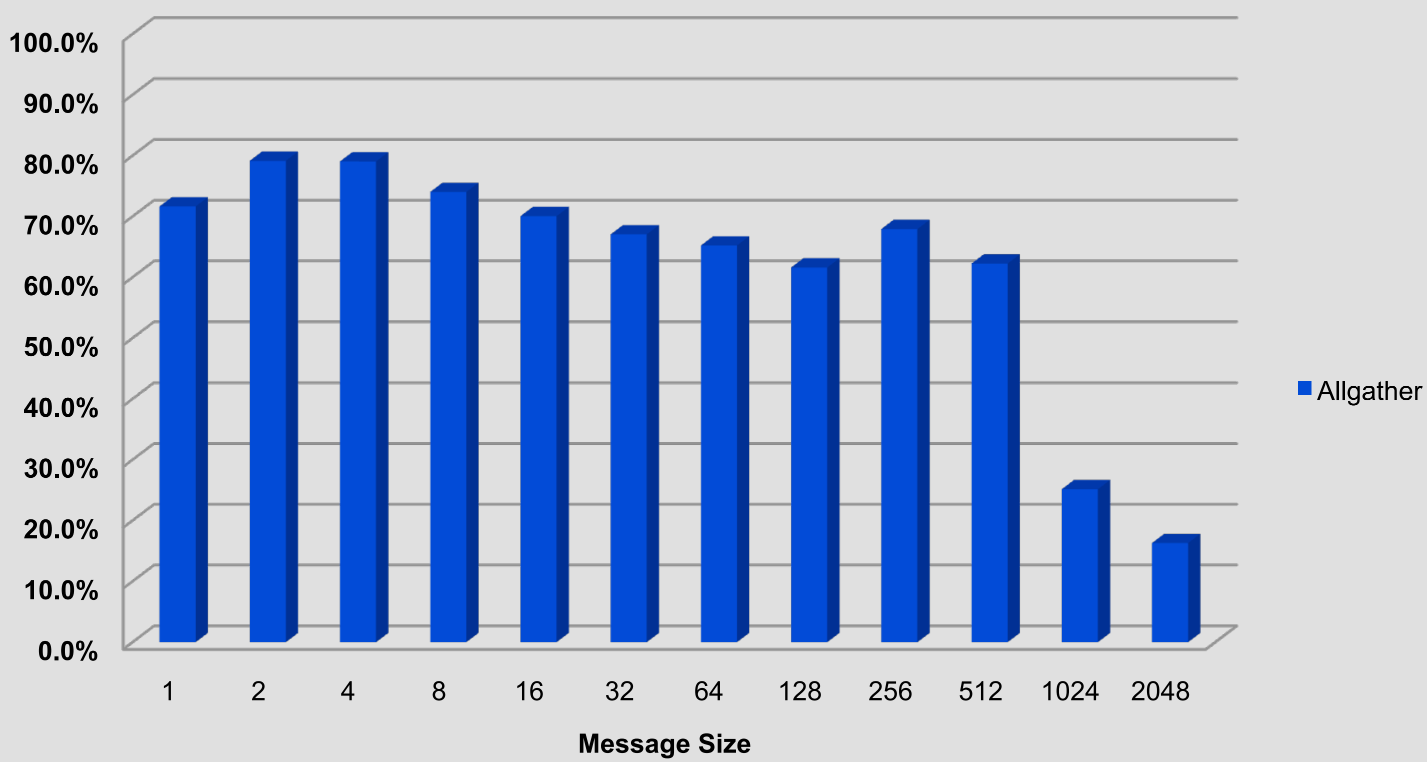
Default vs MPICH_COLL_OPT_OFF=MPI_Reduce



Performance Comparison of MPI_Allgather

Default vs MPICH_COLL_OPT_OFF=MPI_Allgather

Percent Improvement of Optimized MPI_Allgather (compared to MPICH2 algorithm) 1024 PEs on an Istanbul System



Short Message Eager Protocol

- Sender "pushes" message to receiver
  - Sender assumes receiver can handle message and blindly transmits
- If matching receive is posted, receiver
  - routes incoming data directly into specified receive buffer
  - posts notification event to other event queue
- If no matching receive is posted, receiver
  - routes incoming data into unexpected message buffer
  - posts two events to unexpected event queue
  - copies data into specified receive buffer when matching receive is posted
- Message size <= MPICH_MAX_SHORT_MSG_SIZE bytes

Long Message Rendezvous Protocol

- Receiver "pulls" message from sender
- Sender notifies receiver about waiting message via a small header packet
- Receiver requests message from sender after matching receive is posted
- Receiver routes incoming data directly into specified receive buffer
- Message size > MPICH_MAX_SHORT_MSG_SIZE bytes

Long Message Eager Protocol

- Sender assumes receiver will handle message appropriately or will request retransmission
  - Sender blindly transmits data to receiver
- If matching receive is posted, receiver
  - routes incoming data directly into specified receive buffer
  - sends completion acknowledgement to sender
- If no matching receive is posted, receiver
  - creates a long protocol match entry
  - requests retransmission when matching receive is posted
  - routes incoming data directly into specified receive buffer
- Enabled using MPICH_PTLS_EAGER_LONG
- CAUTION: blocking sends and unexpected messages

Configuration

MPI Environment Variables

- Many environment variables available to tune MPI performance
  - Well documented on the MPI man page – Read it!
  - Default settings generally focus on attaining the best performance for most codes – not necessarily your code!
- The MPI environment can change between MPT versions
  - Read the MPI man page and Cray documentation!
- MPICH_ENV_DISPLAY – set to display the MPI environment during MPI initialization
- MPICH_VERSION_DISPLAY - set to display the version of Cray MPT during MPI initialization

Auto-Scaling MPI Environment Variables

- Key MPI variables change their default values depending on job size (total number of ranks)
  - MPICH_MAX_SHORT_MSG_SIZE – threshold for short message eager protocol
  - MPICH_PTL_UNEX_EVENTS – number of entries in unexpected event queue
  - MPICH_UNEX_BUFFER_SIZE – buffer space available for unexpected messages
  - MPICH_PTL_OTHER_EVENTS – number of entries in other event queue (send-side and expected events)
- Users can override defaults with environment variables
- Fine-tuning these variables may help performance
- MPI errors due to insufficiencies indicate which variables need to be increased

Auto-Scaling MPI Environment Variables

- **Default values for various MPI job sizes:**

MPI Environment Variable Name	1,000 PEs	10,000 PEs	50,000 PEs	100,000 PEs
MPICH_MAX_SHORT_MSG_SIZE (This size determines whether the message uses the Eager or Rendezvous protocol)	128,000 B	20,480	4096	2048
MPICH_UNEX_BUFFER_SIZE (The buffer allocated to hold the unexpected Eager data)	60 MB	60 MB	150 MB	260 MB
MPICH_PTL_UNEX_EVENTS (Portals generates <u>two</u> events for each unexpected message received)	20,480 events	22,000	110,000	220,000
MPICH_PTL_OTHER_EVENTS (Portals send-side and expected events)	2048 events	2500	12,500	25,000

MPT Environment Variables – Portals

- MPICH_PTL_MATCH_OFF – set to disable registration of receive requests within portals
  - Allows MPI to perform message matching for the portals device
  - May be beneficial when an application exhausts internal portals resources or when running latency-sensitive applications
- MPICH_PTL_SEND_CREDITS – enables flow control to prevent the Portals event queue from being overrun
  - Value of -1 should prevent queue overflow in any situation
  - Only be used as needed – flow control negatively impacts performance

MPT Environment Variables – Portals

- MPICH_PTL_MEMD_LIMIT – maximum number of Portals Matching Entries (MEs) and Message Descriptors (MDs)
  - May need to increase if pre-posting more than 2048 MPI receives
  - Increase if abort with PtlMEMDPost() failed: PTL_NO_SPACE
  - Default: 2048 Minimum: 2048 Maximum: 65534
  - If you increase MPICH_PTL_MEMD_LIMIT, also increase MPICH_PTL_OTHER_EVENTS to the same limit

Environment Variables MPICH_SMP_OFF

- If set, disable the on-node SMP device and use the Portals device for all MPI message transfers
- Use in a rare cases where code benefits from using Portals matching instead of MPI matching.
- Default: Not enabled.
- Useful for debugging reproducibility issues.

Environment Variables MPICH_FAST_MEMCPY

- If set, enables an optimized memcpy routine in MPI. The optimized routine is used for local memory copies in the point-to-point and collective MPI operations.
  - This can help performance of some collectives that send large (256K and greater) messages.
    - Collectives are almost always faster
    - Speedup varies by message size
    - Example: If message sizes are known to be greater than 1 megabyte, then an optimized memcpy can be used that works well for larges sizes, but may not work well for smaller sizes.
  - Default is not enabled (because there are a few cases that experience performance degradation)
  - Ex: PHASTA at 2048 processes: reduction from 262 s to 195 s

Environment Variables MPICH_COLL_SYNC

- If set, a Barrier is performed at the beginning of each specified MPI collective function. This forces all processes participating in that collective to sync up before the collective can begin.
  - To enable this feature for all MPI collectives, set the value to 1. *Default is off.*
- Can be enabled for a selected list of MPI collectives
- There are *rare* examples where this helps
  - If the code has lots of collectives and MPI profiling shows imbalance (lots of sync time), this *may* help
  - Ex: PHASTA (CFD-turbulent flows) many MPI_Allreduce calls
    - At 2048 processes : reduction from 262 sec to 218 sec.
  - Ex: But slowed down NekTarG (CFD-Blood Flow) by about 7%

Input/Output

- Sometimes I/O causes scalability issues
  - For example, cleaning up some writes improved weak scaling of the CFD code NektarG from 70% to 95% at 1K to 8K cores
- Set file striping appropriately
  - The default stripe count will almost always be suboptimal
  - The default stripe size is usually fine.
  - Once a file is written, the striping information is set
    - Stripe input directories before staging data
    - Stripe output directories before writing data
  - Stripe for your I/O pattern
    - Many-many – narrow stripes
    - Many-one – wide stripes
- Reduce output to stdout
  - Remove debugging reports (e.g. “Hello from rank n of N”)

Environment Variables MPICH_MPIIO_HINTS

- **If set, overrides the default value of one or more MPI-IO hints. This also overrides any value set in the application code with calls to the MPI_Info_set routine.**
- **Hints are applied to the file when it is opened with an MPI_File_open() call.**
- **MPICH_MPIIO_HINTS_DISPLAY**
  - **If set, causes rank 0 in the participating communicator to display the names and values of all MPI-IO hints that are set for the file being opened with the MPI_File_open call.**

Default settings:

```
PE 0:   MPIIO hints for
  c2F.TILT3d.hdf5:

    cb_buffer_size          = 16777216
    romio_cb_read           = automatic
    romio_cb_write          = automatic
    cb_nodes                = #nodes/8
    romio_no_indep_rw       = false
    ind_rd_buffer_size      = 4194304
    ind_wr_buffer_size      = 524288
    romio_ds_read           = automatic
    romio_ds_write          = automatic
    direct_io               = false
    cb_config_list          = *:1
```

Environment Variables MPICH_MPIIO_HINTS (cont.)

Examples:

- **Syntax**
  - `export MPICH_MPIIO_HINTS=data.hdf5:direct_io=true`
- **For FlashIO at 5000 processes writing out 500MB per MPI thread, the following improved performance:**

  ```
  romio_cb_write = "ENABLE"
  romio_cb_read = "ENABLE"
  cb_buffer_size = 32M
  ```

  - When enabled, all collective reads/writes will use collective buffering. When disabled, all collective reads/writes will be serviced with individual operations by each process. When set to automatic, ROMIO will use heuristics to determine when to enable the optimization.
- **For S3D at 10K cores:**

  `romio_ds_write = 'disable'` **- specifies if data sieving is to be done on read.**

  Data sieving is a technique for efficiently accessing noncontiguous regions of data

  `romio_no_indep_rw = 'true'` **- specifies whether deferred open is used.**

  - Romio docs say that this indicates no independent read or write operations will be performed. This can be used to limit the number of processes that open the file.

MPI-IO Improvements

- MPI-IO collective buffering
  - **MPICH_MPIIO_CB_ALIGN=0**
    - Divides the I/O workload equally among all aggregators
    - Inefficient if multiple aggregators reference the same physical I/O block
    - Default setting in MPT 3.2 and prior versions
  - **MPICH_MPIIO_CB_ALIGN=1**
    - Divides the I/O workload up among the aggregators based on physical I/O boundaries and the size of the I/O request
    - Allows only one aggregator access to any stripe on a single I/O call
    - Available in MPT 3.1
  - **MPICH_MPIIO_CB_ALIGN=2**
    - Divides the I/O workload into Lustre stripe-sized groups and assigns them to aggregators
    - Persistent across multiple I/O calls, so each aggregator always accesses the same set of stripes and no other aggregator accesses those stripes
    - Minimizes Lustre file system lock contention
    - Default setting in MPT 3.3

Rank Placement

- In some cases, changing how the processes are laid out on the machine may affect performance by relieving synchronization/ imbalance time.
- The default is currently SMP-style placement. This means that for a multi-node core, sequential MPI ranks are placed on the same node.
  - In general, MPI codes perform better using SMP placement - Nearest neighbor
  - Collectives have been optimized to be SMP aware
- For example, a 12-process job launched on a XT5 node with 2 hex-core processors would be placed as:

PROCESSOR	0	1
RANK	0,1,2,3,4,5	6,7,8,9,10,11

Rank Placement

- The default ordering can be changed using the following environment variable:

  MPICH_RANK_REORDER_METHOD

- These are the different values that you can set it to:

  0: Round-robin placement – Sequential ranks are placed on the next node in the list. Placement starts over with the first node upon reaching the end of the list.

  1: SMP-style placement – Sequential ranks fill up each node before moving to the next.

  2: Folded rank placement – Similar to round-robin placement except that each pass over the node list is in the opposite direction of the previous pass.

  3: Custom ordering. The ordering is specified in a file named MPICH_RANK_ORDER.

- When is this useful?
  - Point-to-point communication consumes a significant fraction of program time and a load imbalance detected
  - Also shown to help for collectives (alltoall) on subcommunicators (GYRO)
  - Spread out IO across nodes (POP)

Rank Order and CrayPAT

- One can also use the CrayPat performance measurement tools to generate a suggested custom ordering.
  - Available if MPI functions traced (-g mpi or –O apa)
  - pat_build –O apa my_program
    - see Examples section of pat_build man page
- pat_report options:
  - mpi_sm_rank_order
    - Uses message data from tracing MPI to generate suggested MPI rank order. Requires the program to be instrumented using the pat_build -g mpi option.
  - mpi_rank_order
    - Uses time in user functions, or alternatively, any other metric specified by using the -s mro_metric options, to generate suggested MPI rank order.

Reordering Workflow

- module load xt-craypat
- Rebuild your code
- pat_build –O apa a.out
- Run a.out+pat
- pat_report –Ompi_sm_rank_order a.out+pat+…sdt/ > pat.report
- Creates MPICH_RANK_REORDER_METHOD.x file
- Then set env var MPICH_RANK_REORDER_METHOD=3 AND
- Link the file MPICH_RANK_ORDER.x to MPICH_RANK_ORDER
- Rerun code

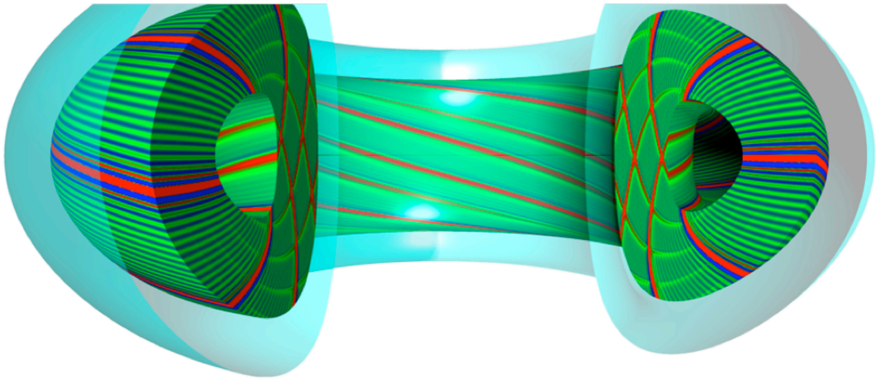
CrayPAT example

```
Table 1:  Suggested MPI Rank Order

 Eight cores per node:  USER Samp per node
 Rank         Max     Max/         Avg     Avg/   Max Node
Order   USER Samp      SMP  USER Samp      SMP   Ranks
    d       17062    97.6%      16907   100.0%   832,328,820,797,113,478,898,600
    2       17213    98.4%      16907   100.0%   53,202,309,458,565,714,821,970
    0       17282    98.8%      16907   100.0%   53,181,309,437,565,693,821,949
    1       17489   100.0%      16907   100.0%   0,1,2,3,4,5,6,7
```

- This suggests that
  1. the custom ordering “d” might be the best
  2. Folded-rank next best
  3. Round-robin 3rd best
  4. Default ordering last

Reordering example GYRO



- GYRO 8.0
  - B3-GTC problem with 1024 processes
- Run with alternate MPI orderings
  - Custom: profiled with with –O apa and used reordering file MPICH_RANK_REORDER.d

Reorder method	Comm. time
Default	11.26s
0 – round-robin	6.94s
2 – folded-rank	6.68s
d-custom from apa	8.03s

CrayPAT suggestion almost right!

Reordering example
TGYRO

- TGYRO 1.0
  - Steady state turbulent transport code using GYRO, NEO, TGLF components
- ASTRA test case
  - Tested MPI orderings at large scale
  - Originally testing weak-scaling, but found reordering very useful

Reorder method	TGYRO wall time (min)		
	20480	40960	81920
Default	99m	104m	105m
Round-robin	66m	63m	72m

Huge win!

Tips & Recommendations

Cray MPT – General Tips

- Always use the compiler wrappers to compile!
  - Always specify the compiler wrappers when running configure!
- Use a recent version of MPT (current 5.2.0)
  - Significant improvements (e.g. allgatherv in 4.0.0 and later)
- Update environment variables for new versions of MPT
  - Updated algorithms might have different requirements
  - Current versions attempt to set the right buffer sizes at launch based on job size rather than using static settings
  - Suggestion: if you use env vars based on previous versions, try using recent verisons w/o env vars
- Status: Kraken: default 5.0.0 JaguarPF: default 4.0.0

Cray MPT – Messaging Tips

- Performs best when every message is expected prior to receipt, but ensuring such can be difficult or impossible
- Special handling of unexpected messages for both MPI and Portals to maximize performance and scalability
- Excessively bad application behavior can exhaust available resources for handling unexpected messages and events, resulting in application termination.
  - Short term fix: allocate additional resources via environment variables
  - Long term fix: modify application to improve communication behavior

Portals Errors

Error	Description / Cause	Suggested Fix
`PTL_PT_NO_ENTRY`	Memory mapping error / improper stack initialization	Request refund and resubmit job
`PTL_NAL_FAILED`	Network layer error / node or network failure	Request refund and resubmit job
`PTL_EQ_DROPPED`	Event dropped from queue / insufficient space in queue	Increase resources with environment variables, change application communication profile
`PTL_SEGV`	Invalid user address supplied to portals	Fix invalid pointers in application code
`PTL_PT_VAL_FAILED`	Invalid address / invalid buffer parameter in MPI	Fix invalid pointers in application code (MPI)
`PTL_NO_SPACE`	Insufficient memory for internal buffers	Reduce app. memory, increase MPICH_PTL_MEMD_LIMIT, set MPICH_PTL_MATCH_OFF

Step by Step

1. Fix any load imbalance – consider decomposition and rank order
2. Fix your hotspots
   1. Communication
      - Pre-post receives
      - Overlap computation and communication
      - Reduce collectives
      - Adjust MPI environment variables
      - Use rank reordering
   2. Computation
      - Examine the hardware counters and compiler feedback
      - Adjust the compiler flags, directives, or code structure to improve performance
   3. I/O
      - Stripe files/directories appropriately
      - Use methods that scale
        - MPI-IO or Subsetting

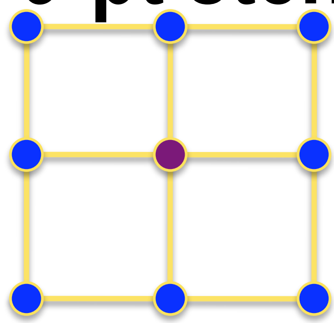
At each step, check your *answers* **and** *performance*.

Between each step, gather your data again.

MPI Programming Techniques
Pre-posting receives

- If possible, pre-post receives before the matching sends
  - Optimization technique for all MPICH installations (not just MPT)
  - Not sufficient to simply put receive immediately before send
  - Put significant amount of computation between receive-send pair
- Do not go crazy pre-posting receives. You can (and will) overrun the resources available to Portals.
- Code example
  - Halo update – with four buffers (n,s,e,w), post all receive requests as early as possible. Makes a big difference on CNL (not as important on Catamount).

MPI Programming Techniques
Example: 9-pt stencil pseudo-code



Basic

```
9-pt computation

Update ghost cell
 boundaries
  East/West IRECV,
   ISEND, WAITALL
  North/South IRECV,
   ISEND, WAITALL
```

Maximal Irecv preposting

```
Prepost all IRECV

9-pt computation

Update ghost cell
 boundaries
  East/West ISEND,
   Wait on E/W IRECV
   only
  North/South ISEND,
   Wait on the rest
```

*Makes use of temporary buffers

MPI Programming Techniques
Overlapping communication with computation

- Use non-blocking send/recvs to overlap communication with computation whenever possible
  - Typical pattern:
    1. Pre-post non-blocking receive
    2. Compute a “reasonable” amount to ensure effective pre-posting
    3. Post non-blocking send
    4. Compute as much as possible to maximize overlap of comm. and comp.
    5. Wait on communication to finish only when absolutely necessary

MPI Programming Techniques
Overlapping communication with computation

- In some cases, it may be better to replace collective operations with point-to-point communications to overlap communication with computation
  - Caution: Do not blindly reprogram every collective by hand
  - Concentrate on the parts of your algorithm with significant amounts of computation that can overlap with the point-to-point communications when a [blocking] collective is replaced

MPI Programming Techniques
Reduce Collective Communications

- Avoid using collective communications whenever possible
  - MPI collectives are blocking, leading to large sync times
  - Collective communication can cripple scalability
- Use algorithms that only require local info when possible
  - Consider duplicating computation to reduce communication
- When an algorithm must communicate “globally”:
  - Use MPT collectives that have been optimized by Cray
  - Minimize the scope of the collective operation
  - Minimize the number of collectives through aggregation
  - Consider implementing a non-blocking collective only if justified after careful analysis

MPI Programming Techniques
Aggregating data

- For very small buffers, aggregate data into fewer MPI calls (especially for collectives)
  - 1 all-to-all with an array of 3 reals is clearly better than 3 all-to-alls with 1 real
  - Do not aggregate too much. The MPI protocol switches from a short (eager) protocol to a long message protocol using a receiver pull method once the message is larger than the eager limit. This limit is by default 128000 bytes, but it can be changed with the MPICH_MAX_SHORT_MSG_SIZE environment variable. The optimal size for messages most of the time is less than the eager limit.
- Example – DNS
  - Turbulence code (DNS) replaced 3 AllGatherv's by one with a larger message resulting in 25% less runtime for one routine

MPI Programming Techniques
Aggregating data: Example from CFD

Original

```
for (index = 0; index < No; index++){
   double tmp;
   tmp = 0.0;
   out_area[index] = Bndry_Area_out(A,
labels[index]);
   gdsum(&outlet_area[index],1,&tmp);
}
for (index = 0; index < Ni; index++){
  double tmp;
  tmp = 0.0;
  in_area[index] = Bndry_Area_in(A,
labels[index]);
  gdsum(&inlet_area[index],1,&tmp);
}


void gdsum (double *x, int n, double *work)
   {
     register int i;
     MPI_Allreduce (x, work, n, MPI_DOUBLE,
   MPI_SUM, MPI_COMM_WORLD);
     /* *x = *work; */
     dcopy(n,work,1,x,1);
     return;
   }
```

Improved

```
   for (index = 0; index < No; index++){
      out_area[index] = Bndry_Area_out(A,
labels[index]);
   }

   /* Get gdsum out of for loop */
   tmp = new double[No];
   gdsum (outlet_area, No, tmp);
   delete tmp;

   for (index = 0; index < Nin; index++){
     in_area[index] = Bndry_Area_in(A,
labels[index]);
   }

   /*  Get gdsum out of for loop */
   tmp = new double[Ni];
   gdsum(inlet_area, Ni, tmp);
   delete tmp;
```

Hybridization

OpenMP

- When does it pay to add/use OpenMP in my MPI code?
  - Add/use OpenMP when code is network bound
  - As collective and/or point-to-point time increasingly becomes a problem, use threading to keep number of MPI processes per node to a minimum
  - Be careful adding OpenMP to memory bound codes – can hurt performance
  - Be careful to match memory affinity to thread affinity
    - Pre-touch memory from correct thread after allocation
  - It is code/situation dependent!
  - Consider one MPI process on each CPU and one OpenMP thread per available core within each process
    - Often gives results almost as good as a fully optimized one-process-per-node code (with OpenMP threads across all of the cores on the node) with significantly less development overhead

OpenMP
aprun depth

- Must get "aprun –d" correct
  - -d (depth) Specifies the number of threads (cores) for each process. ALPS allocates the number of cores equal to depth times processes.
  - The default depth is 1. This option is used in conjunction with the OMP_NUM_THREADS environment variable.
  - Also used to get more memory per process
    - Get 1 or 2 GB limit by default (machine dependent)
  - Many have gotten this wrong, so it is important to understand how to use it properly!
    - If you do not do it correctly, a hybrid OpenMP/MPI code can get multiple threads spawned on the same core which can be disastrous.

OpenMP
aprun depth (cont.)

All on core 0
All on 1 node

One thread per core as desired!!!

% setenv OMP_NUM_THREADS 4

% aprun -n 4 -q ./omp1 | sort

```
Hello from rank 0, thread 0, on nid00291. (core affinity = 0)
Hello from rank 0, thread 1, on nid00291. (core affinity = 0)
Hello from rank 0, thread 2, on nid00291. (core affinity = 0)
Hello from rank 0, thread 3, on nid00291. (core affinity = 0)
Hello from rank 1, thread 0, on nid00291. (core affinity = 1)
Hello from rank 1, thread 1, on nid00291. (core affinity = 1)
Hello from rank 1, thread 2, on nid00291. (core affinity = 1)
Hello from rank 1, thread 3, on nid00291. (core affinity = 1)
Hello from rank 2, thread 0, on nid00291. (core affinity = 2)
Hello from rank 2, thread 1, on nid00291. (core affinity = 2)
Hello from rank 2, thread 2, on nid00291. (core affinity = 2)
Hello from rank 2, thread 3, on nid00291. (core affinity = 2)
Hello from rank 3, thread 0, on nid00291. (core affinity = 3)
Hello from rank 3, thread 1, on nid00291. (core affinity = 3)
Hello from rank 3, thread 2, on nid00291. (core affinity = 3)
Hello from rank 3, thread 3, on nid00291. (core affinity = 3)
```

% setenv OMP_NUM_THREADS 4

% aprun -n 4 -d 4 -q ./omp | sort

```
Hello from rank 0, thread 0, on nid00291. (core affinity = 0)
Hello from rank 0, thread 1, on nid00291. (core affinity = 1)
Hello from rank 0, thread 2, on nid00291. (core affinity = 2)
Hello from rank 0, thread 3, on nid00291. (core affinity = 3)
Hello from rank 1, thread 0, on nid00291. (core affinity = 4)
Hello from rank 1, thread 1, on nid00291. (core affinity = 5)
Hello from rank 1, thread 2, on nid00291. (core affinity = 6)
Hello from rank 1, thread 3, on nid00291. (core affinity = 7)
Hello from rank 2, thread 0, on nid00292. (core affinity = 0)
Hello from rank 2, thread 1, on nid00292. (core affinity = 1)
Hello from rank 2, thread 2, on nid00292. (core affinity = 2)
Hello from rank 2, thread 3, on nid00292. (core affinity = 3)
Hello from rank 3, thread 0, on nid00292. (core affinity = 4)
Hello from rank 3, thread 1, on nid00292. (core affinity = 5)
Hello from rank 3, thread 2, on nid00292. (core affinity = 6)
Hello from rank 3, thread 3, on nid00292. (core affinity = 7)
```

OpenMP – Scope all variables!

```
int i, j, k;

#pragma omp parallel shared(t, new, old,
 nrl, dt, NR, NC, NITER) private(d)

 #pragma omp for schedule(runtime) nowait
    for (i = 2; i <= nrl-1; i++)
      for (j = 1; j <= NC; j++){
        t[*new][i][j] = 0.25 *
        (t[old][i+1][j] + t[old][i-1][j] +
        t[old][i][j+1] + t[old][i][j-1]);
        d = MAX(fabs(t[*new][i][j] -
                t[old][i][j]), d);
```

```
int i, j, k;

#pragma omp parallel shared(t, new, old, nrl,
  dt, NR, NC, NITER) private(d,i,j)

 #pragma omp for schedule(runtime) nowait
    for (i = 2; i <= nrl-1; i++)
      for (j = 1; j <= NC; j++){
        t[*new][i][j] = 0.25 *
        (t[old][i+1][j] + t[old][i-1][j] +
        t[old][i][j+1] + t[old][i][j-1]);
        d = MAX(fabs(t[*new][i][j] -
                t[old][i][j]), d);
```

In this particular case, the homb benchmark got wrong answers and failed to scale well when using PGI and Pathscale.

Closing Remarks

Last words

- MPT provides optimized, high-performance communication
  - Sometimes requires guidance and tuning – also patience and perseverance
- Environment variables are an easy way to improve performance
  - Familiarize yourself with 'man mpi' and remain up-to-date
- The is no replacement for good MPI programming practices
  - Pre-posting receives, overlap computation and communication, reduce collective communications, aggregate data for communication
- Rank reordering can significantly improve performance
- Use depth option to aprun with OpenMP
- Remember your parallel I/O – it can be crippling
- Some of this may not show a benefit at <1K processes, but it can reap huge gains at 10K to 100K processes
- Thanks to Jeff Larkin of Cray for permission to use his slides